An Efficient FPGA-based Floating Random Walk Solver for Capacitance Extraction using SDAccel

Xin Wei[1], Changhao Yan*[1], Hai Zhou[1,2], Dian Zhou[1,3], Xuan Zeng*[1]
[1]State Key Lab of ASIC and System, Microelectronics Department, Fudan University, China
[2]Department of Electrical Engineering and Computer Science, Northwestern University, USA
[3]Department of Electrical Engineering, University of Texas at Dallas, USA

***Abstract*—The floating random walk (FRW) algorithm is an important method widely used in the capacitance extraction of very large-scale integration (VLSI) interconnects. FRW could be both time-consuming and power-consuming as the circuit scale grows. However, its highly parallel nature prompts us to accelerate it with FPGAs, which have shown great performance and energy efficiency potential to other computing architectures. In this paper, we propose a scalable FPGA/CPU heterogeneous framework of FRW using SDAccel. Large-scale circuits are partitioned first by the CPU into several segments, and these segments are then sent to the FPGA random walking one by one. The framework solves the challenge of limited FPGA on-chip resource and integrates both merits of FPGAs and CPUs by targeting separate parts of the algorithm to suitable architecture, and the FPGA bitstream is built once for all. Several kernel optimization strategies are used to maximize performance of FPGAs. Besides, the FRW algorithm we use is the naive version with walking on spheres (WOS), which is much simpler and easier to implement than the complicatedly optimized version with walking on cubes (WOC). The implementation on AWS EC2 F1 (Xilinx VU9P FPGA) shows up to 6.1x performance and 42.6x energy efficiency over a quad-core CPU, and 5.2x energy efficiency over the state-of-the-art WOC implementation on an 8-core CPU.**

I. Introduction

With the development of nanometer VLSI circuits, the metal line width decreases gradually. Meanwhile, the total length and number of layers of interconnects increase due to the expansion of chip size, resulting in increased ratio of the delay caused by interconnect parasitic capacitance to the total delay. In this context, it has become an important issue to efficiently extract interconnect capacitance.

One kind of field-solver algorithm for capacitance extraction is based on deterministic algorithms such as boundary element method (BEM) [1–3] and finite element method (FEM) [4, 5]. The other kind is the floating random walk (FRW) algorithm based on Monte Carlo method. Compared with the deterministic algorithms, FRW has advantages of intrinsic parallelism, good scalability and low memory occupancy, which make it fit for large-scale integrated circuits. The 2-D FRW algorithm for capacitance extraction was first proposed in 1992 [6]. Its basic idea is to approximate the exact value of Gauss Theorem integral expression by statistical sampling. According to the shape of transition domain, FRW can be categorized into walking on spheres (WOS) and walking on cubes (WOC). WOC, where the transition domain is cubic, is now widely used because it takes fewer steps to hit a cuboid conductor [7–10], but it requires sophisticated pretreatment of Green's function tables and weight value tables, and optimizations such as space management [8]. The FRW algorithm is computationally intensive and it would consume much time and energy to deal with large-scale structures. Fortunately, thanks to the intrinsic parallelism of FRW, parallel architectures like multi-core CPUs, GPUs and field-programmable gate arrays (FPGAs) may be leveraged to accelerate it. So far, the FRW algorithm has been implemented on multi-core CPUs [7, 8] and GPUs [9, 10], but it has not been implemented on FPGAs.

The FPGA is originally developed as an ASIC verification tool. In recent years, it has been increasingly used in algorithm acceleration because of its high parallelism and flexible hardware configuration. Compared with the GPU, FPGA has an absolute advantage in low power scenarios. In addition, on the basis of Hardware Description Language (HDL), the development of High Level Synthesis (HLS) has greatly improved FPGA's development efficiency, bringing it broader application prospects. With the help of Xilinx SDAccel toolchain [11], not only heterogeneous computing of FPGA/CPU can be conveniently implemented, but the development cycle is also shortened compared with traditional RTL design flow. Research have demonstrated efficiency of FPGAs with SDAccel for implementations of neural networks, N-Body simulation, frequent itemset mining, finite-difference time domain method [12–15].

Challenges in implementing FRW on FPGAs are limited on-chip resource and non-intuitive kernel design, which are the main focus of this paper. The rest of paper is organized as follows. Section II is a brief introduction and formula derivation of the FRW algorithm with walking on spheres. In Section III, for the first time we propose an FPGA/CPU heterogeneous framework of FRW, and kernel optimization techniques are detailed. We implement our design on Amazon Web Services (AWS) cloud, and experimental results show both advantage of performance and energy efficiency over CPUs in Section IV. Notably, even with the naive WOS algorithm and short programs, our implementation has better energy efficiency than complicatedly optimized WOC. Finally, Section V concludes this work.

II. Background

Consider the electric potential of point $\mathbf{r}$ in a homogeneous dielectric medium:

$$\phi(\mathbf{r}) = \oint_S P(\mathbf{r}, \mathbf{r}_1)\phi(\mathbf{r}_1)ds_1 \quad (1)$$

* Corresponding authors: {yanch, xzeng}@fudan.edu.cn

978-3-9819263-2-3/DATE19/©2019 EDAA

where S is a closed surface with $\mathbf{r}$ inside. The surface Green's function $P(\mathbf{r}, \mathbf{r}_1)$ indicates the probability density function (PDF) of arbitrary point $\mathbf{r}_1$ out of S jumping from fixed $\mathbf{r}$. Actually, this means $\phi(\mathbf{r})$ equals the average potential of points on its enveloping surface S, and we can have a good estimation of it by sampling enough points on S.

However, $\phi(\mathbf{r}_1)$ is also unknown in most cases. We can recursively implement (1) until a known potential $\phi(\mathbf{r}_k)$:

$$\phi(\mathbf{r}) = \oint_{S_1} P(\mathbf{r}, \mathbf{r}_1) \oint_{S_2} P(\mathbf{r}_1, \mathbf{r}_2) \cdots \oint_{S_k} P(\mathbf{r}_{k-1}, \mathbf{r}_k) \phi(\mathbf{r}_k) ds_k \cdots ds_2 ds_1 \tag{2}$$

where closed surface $S_i (i = 1, 2, \cdots, k)$ envelops point $\mathbf{r}_i (i = 1, 2, \cdots, k)$ and $P(\mathbf{r}_{i-1}, \mathbf{r}_i) (i = 1, 2, \cdots, k)$ is the surface Green's function for fixed $\mathbf{r}$. As illustrated in Fig. 1, floating random walk on spheres (WOS) is a series of hops of spherical transition domain. At each hop, a conductor-free sphere with maximum radius is constructed centering $\mathbf{r}_{i-1}$. In the meantime, a random point $\mathbf{r}_i$ on that sphere surface is then chosen by $P(\mathbf{r}_{i-1}, \mathbf{r}_i)$. Clearly, $P(\mathbf{r}_{i-1}, \mathbf{r}_i)$ is only dependent on the radius $|\mathbf{r}_{i-1} - \mathbf{r}_i|$ here for sphere enveloping surfaces. The walk ends on the moment $\mathbf{r}_i$ hitting a conductor surface ($\varepsilon = 10^{-5}$), which means $\phi(\mathbf{r}_i)$ is already known.

Now that the potential of each point can be obtained, it is straightforward to extract capacitances of interconnects. According to the Gauss Theorem,

$$\begin{aligned} Q_i &= \oint_{G_i} \mathbf{D}(\mathbf{r}) \cdot \mathbf{n}(\mathbf{r}) ds \\ &= \oint_{G_i} \varepsilon_r \varepsilon_0 (- \bigtriangledown \phi(\mathbf{r})) \cdot \mathbf{n}(\mathbf{r}) ds \end{aligned} \tag{3}$$

where G_i is a Gaussian surface enclosing conductor i and its charge Q_i can be derived with $\phi(\mathbf{r})$. $\mathbf{n}(\mathbf{r})$ is the exterior normal direction of G_i at point $\mathbf{r}$. Substitute (1) into (3) and under the condition of walking on spheres, we have

$$Q_i = \oint_{G_i} \frac{1}{A_{G_i}} \oint_{S_1} \omega(\mathbf{r}, \mathbf{r}_1) P(\mathbf{r}, \mathbf{r}_1) \phi(\mathbf{r}_1) ds_1 ds \tag{4}$$

where A_{G_i} is the surface area of G_i and the weight value

$$\omega(\mathbf{r}, \mathbf{r}_1) = -2\varepsilon_r \varepsilon_0 A_{G_i} \frac{\mathbf{n}(\mathbf{r}) \cdot (\mathbf{r}_1 - \mathbf{r})}{|\mathbf{r}_1 - \mathbf{r}|^2} \tag{5}$$

There are two integrals in (4). For the first integral, we sample N_{Start} random starting points on G_i; for the second one, $N_{PathPerStart}$ random points are sampled on S_1, which makes $N_{Path} = N_{Start} * N_{PathPerStart}$ paths in total. Then the formulae can be calculated as (1) indicates and the mean value of their answers would be a great approximation to the true value providing sufficient paths.

It can be seen from above that the FRW algorithm is very compute-intensive especially when dealing with large-scale circuits, which may cost both a lot of time and energy. However, thanks to the intrinsic high parallelism of FRW, it can be accelerated by GPUs or FPGAs. So far, the algorithm has been implemented on GPUs [9, 10]. In the next section, we present a novel FPGA/CPU heterogeneous framework for the algorithm.

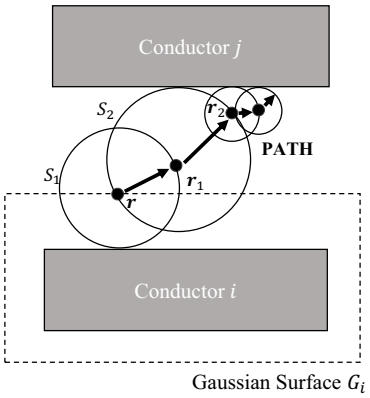

Fig. 1. Illustration of floating random walk (FRW).

III. Heterogeneous Framework and Kernel Optimization

In this section, the FPGA/CPU heterogeneous framework is first illustrated, where the input of GDS files is addressed in the CPU and the FPGA deals with hops of FRW. As we all know, the generation of FPGA bitstreams could cost double-digit hours. But with this framework, the bitstream is built once for all, leaving the pre-process and partition of GDS files completed by CPUs. Several optimization techniques on FPGA kernels are also detailed afterwards.

A. FPGA/CPU Heterogeneous Framework

Noticed that one path in the FRW algorithm is not relevant to any of the others, which means the algorithm is highly parallel and is suitable for FPGA implementation. However, the resource of an FPGA is very limited considering the scale of circuits can be very large nowadays. Hence the partition of circuits is a pre-requisite for FPGA implementation.

As illustrated in Fig. 2, our framework consists of 3 parts:

- In part 1, the CPU reads the GDS file, constructs a suitable Gaussian surface G_i for the master conductor i and samples N_{Start} random starting points on G_i. Then the circuit is partitioned into several segments. The scale of each segment is determined by the resource of the FPGA. And one segment must have some overlay with its neighbor to accommodate the paths which hop out of its local segment. This would be detailed in the next subsection.
- Part 2 is in the charge of an FPGA. The FPGA first reads conductor information and starting points' coordinates of segment i from the CPU. After specified maximum steps of hops, any of the paths is either having hit a conductor or having not hit any conductor. For the paths having hit conductor j, the FPGA outputs the number j to the CPU; for the paths having not hit any conductor, the FPGA outputs its current coordinates to the CPU.
- In part 3, the CPU continues to compute the paths which have not hit any conductor in part 2 after all segments have already been processed by the FPGA. As most of the paths would terminate in the local segment in the FPGA, only a

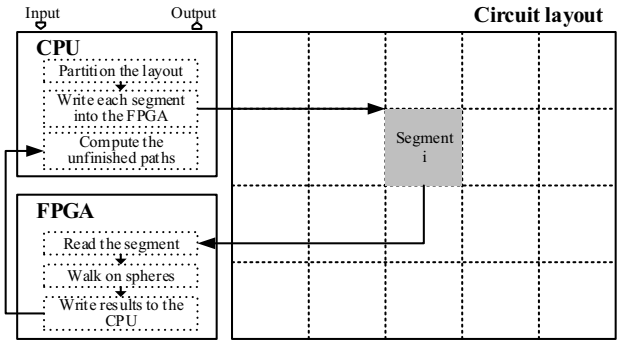

Fig. 2. FPGA/CPU heterogeneous framework for the FRW algorithm.

few paths are left for the CPU and this costs little time. Our experiment shows the ratio of number of paths left for the CPU to the total number can be no more than 0.4% when the circuit scale is large (Table II). The computation and print of the final result is also addressed in this part.

The FPGA/CPU heterogeneous framework presented above has several advantages. 1) The FPGA bitstream is built once and for all to make full use of its resources, while the circuits are read and partitioned by the CPU, which gives much flexibility to scale and shapes of the input circuits. 2) The framework is scalable and may be extended to architectures with multiple FPGAs. 3) The framework integrates both merits of FPGAs and CPUs by targeting separate parts of the algorithm to suitable computing device.

B. Partition Method with Margin

As introduced in subsection A, we have to partition the circuit into several segments before writing them into the FPGA. Actually, not only the circuit is partitioned, the Gaussian surface is also partitioned together. The data transferred from the CPU to the FPGA are conductor information and coordinates of the starting points on the Gaussian surface. The conductor boundary and the Gaussian surface (GS) boundary cannot be the same with the segment boundary (Fig. 3).

- **Segment boundary:** If a path hops out of the segment boundary, it may go back to the current segment or not in its next hop. But we do not have enough information to make this judgement with only conductor information of current segment. This circumstance is called "Out of local segment". When it occurs, the path terminates in the FPGA and waits for computation in the CPU with full circuit information.
- **Conductor boundary:** The conductor boundary is broader than the segment boundary and it is the boundary of conductors to be transferred into the FPGA. When a path is close to the segment boundary, the additional conductor (like the conductors between solid line and dotted line in Fig. 3) can help avoid hopping into the conductors of adjacent segments.
- **GS boundary:** The Gaussian surface boundary is narrower than the segment boundary and it is the boundary of starting points to be transferred into the FPGA. There would be many starting points very close to the segment boundary if the GS boundary and the segment boundary coincide, whose paths are easy to hop out of local segment. So we shrink the GS boundary a little. To make sure any of the starting points is neither duplicated nor neglected, there should not be any overlay or gap between adjacent GS boundaries.

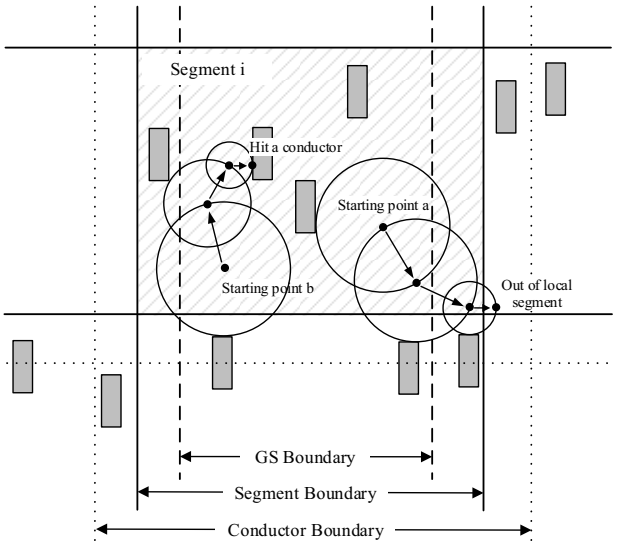

Fig. 3. Partition Method with Margin.

C. FPGA Kernel Optimization

The FPGA is a reconfigurable and inherently parallel processing fabric. And the FPGA kernels should be optimized elaborately to extract maximum parallelism of the FPGA taking its architecture into account.

Algorithm 1 is the original core pseudo-code of the FRW kernel, which has omitted the outer loops of enumerating starting points and groups ($N_{group} = N_{PathPerStart}/N_{thread}$). Here we try to realize concurrent walking of N_{thread} paths. It is always recommended to prefetch data from global memory to local memory on the FPGA at the beginning of the kernel, which is also omitted in Algorithm 1.

It is easier for SDAccel compiler to optimize the code when for loops are fixed-length. Hence an upper limit of number of hop steps is set 150. We use the most straightforward method to calculate the radius of each hop. That is computing the distances between current position and each face of all conductors in the local segment, and then selecting the minimum. Optimization techniques are as follows.

a) Pipes streaming data from one kernel to another: It is observed that the generation of random numbers (line 6 in Algorithm 1) is independent and can be isolated as a separate kernel. The OpenCL 2.0 specification has introduced pipes to stream data from one kernel to another, which stores data organized as a FIFO and improves the latency of data transfer between kernels. As illustrated in Fig. 4, we use LCGs (linear congruential generators) to generate random numbers in the PRNG (pseudo random number generator) kernel and

Algorithm 1 Original FRW kernel (partial).

```
1: for each thread do
2:   for each step of hops do
3:     for each conductor do
4:       Calculate the radius
5:     end for
6:     Generate random numbers
7:     if not (Hit conductor or Out of Local Segment) then
8:       Hop according to the radius and random numbers
         calculated
9:     end if
10:  end for
11: end for
```

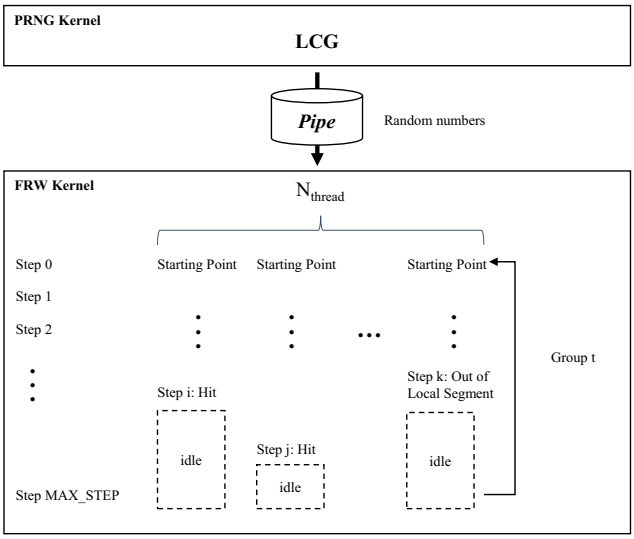

Fig. 4. A pipe streams data from random generator kernel to FRW kernel.

it transfers the random numbers to the FRW kernel through a pipe.

b) Loop unrolling and pipelining: Appropriate loop unrolling and pipelining are the key to the concurrency of FPGA kernels. Constrained by the SDAccel compiler, only the inner-most loop can be unrolled. Hence the loop order is adjusted in Algorithm 2 to meet this condition. In addition, for double nested loops, the compiler would try to unroll the inner loop (line 4 in Algorithm 2) and pipeline the outer loop (line 3 in Algorithm 2) if applying the pipeline attribute $\_\_attribute\_\_((xcl\_pipeline\_loop))$ on the outer loop.

c) Array partition: Usually, the BRAM (block RAM) is implemented to store a relatively large set of data on the FPGA. The BRAMs are dual-port RAMs, which allow for parallel access. But this is not enough compared to the parallelism required (N_{thread}). The array partition attribute $\_\_attribute\_\_((xcl\_array\_partition))$ can help completely partition the BRAM into into distributed RAMs or independent registers, which can be read or written simultaneously.

Algorithm 2 Adjusted loop order for unrolling and pipelining.

```
1: for each step of hops do
2:   __attribute__((xcl_pipeline_loop))
3:   for each conductor do
4:     for each thread do
5:       Calculate the radius
6:     end for
7:   end for
8:   read random numbers from pipe
9:   for each thread do
10:    if not (Hit conductor or Out of Local Segment) then
11:      Hop according to the radius and random numbers
         calculated
12:    end if
13:  end for
14: end for
```

IV. Experimental Results

We implement our FPGA/CPU heterogeneous design with Xilinx SDAccel toolchain [11] on the Amazon Web Services (AWS). To make a comparison, we also implement the same basic FRW algorithm on a quad-core CPU. Moreover, comparison to the CPU and GPU implementations [8, 9] with optimized FRW algorithm is made. Numerical results show high energy efficiency of the FPGA/CPU heterogeneous framework. And the framework has great potential to be further optimized thanks to its scalable characteristic.

A. Experimental Settings

Here are two experimental platforms.

- **FPGA/CPU:** The platform is an AWS EC2 F1 instance equipped with a Xilinx VU9P FPGA board and an Intel Xeon E5-2686 8-core CPU (2.3GHz). The development environment is Xilinx SDAccel 2017.1, with full-precision (32-bit) floating-point number used. And the power consumption of the FPGA board is provided by Amazon FPGA Image (AFI) Management Tools.
- **CPU:** The platform is a PC with an Intel Core i5-4570 quad-core CPU (3.2GHz). The algorithm is implemented in C++.

Test case 1 is similar to that used in [8] and [9] for comparison (Fig. 5). It consists of 41 wires in 3 metal layers, where 3 parallel wires are on M2 layer and each of M1 and M3 layer has 19 parallel wires. In test case 1, the circuit is partitioned into 4 segments, the number of starting points is 100, and the number of total paths is 102.4k.

To help better measure the performance of our design, test case 2 is designed with more wires on the basis of test case 1. That is extending the 3 wires on M2 layer, and duplicating wires on M1 and M3 layer to make each layer 800 wires. In test case 2, the circuit is partitioned into 20 segments, the number of starting points is 2522, and the number of total paths is 10088k.

TABLE I
VALIDATION OF KERNEL OPTIMIZATION TECHNIQUES WITH CASE 2 ($\#thread$=8)

		Resource				*FPGA performance*		
	Kernel	*FF*	*LUT*	*DSP48E*	*BRAM_18K*	*Runtime(s)*	*#path*	*Performance(paths/s)*
Unoptimized	FRW	33719	31310	132	133	1444.82	10044k	6951.44
+Pipe	FRW	32961	30813	128	131	1433.48	10044k	7006.43
	PRNG	1547	1320	11	2			
+Unroll&Pipeline	FRW	103266	83224	419	123	490.56	10044k	20473.71
	PRNG	5694	2809	74	2			
+Array Partition	FRW	170746	101507	419	120	483.52	10044k	20771.81
	PRNG	14986	10063	116	0			

TABLE II
ANALYSIS OF RESOURCES AND PERFORMANCE WITH INCREASING NUMBER OF THREADS (CASE 2)

#thread	*Resource*				*Freq(MHz)*	*Power(W)*	*Performance(paths/s)*		*% of paths left for CPU*	*Err%*
	FF	*LUT*	*DSP48E*	*BRAM_18K*			*FPGA*	*FPGA+CPU*		
16	291292	186850	1007	120	230	12	18844.92	18190.89	0.4%	0.5%
8	185732	111570	535	120	250	11	20771.81	19957.99	0.4%	0.3%
4	133115	74286	299	120	250	11	15443.75	15013.74	0.4%	0.5%
2	78228	44480	181	112	250	11	10203.82	10027.56	0.4%	0.3%
1	64567	33530	119	104	250	11	6079.357	6028.6	0.4%	0.4%

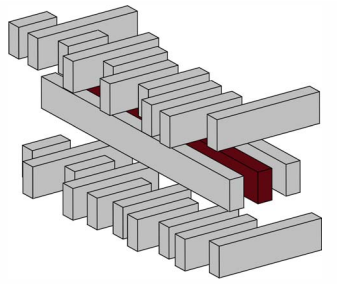

Fig. 5. Illustration of test cases (The master conductor is painted red).

B. Kernel Optimization Analysis

We have introduced three kernel optimization techniques in Section III-C. To validate the effect of theses optimization techniques, we first experiment with the unoptimized kernel, and then add each of the optimizations step by step. The experiment is carried out with test case 2 ($\#thread$=8) and the results are shown in Table I. As the FPGA performance is the only focus here, the data of CPU performance are not listed.

a) Pipe: Before adding a pipe to stream data between kernels, the original unoptimized kernel has been divided into 2 kernels - PRNG kernel and FRW kernel. In this case, the PRNG kernel is relatively simple and costs only a few resources. Consequently, the speedup of performance is small too.

b) Unroll&pipeline: Loop unrolling and pipelining is the key optimization in this case because it greatly improves the parallelism of the kernel. The speedup ratio is 2.9x to the previous kernel. The reason why the speedup ratio is less than the number of thread (8) is that the compiler would do some automatic optimization including loop pipelining even without attributes added. It is also noticed that the resources (FF, LUT and DSP) consumed by the FRW kernel are about 3 times those of the previous kernel.

c) Array partition: Array partition can help improve the concurrency of data reading and writing. We can see from Table I that the number of BRAM consumed decreases and the number of FF consumed increases compared with the previous kernel. The performance is also improved.

C. Performance and Energy Efficiency Analysis

Table II illustrates the resources consumed and the performance of our framework when the number of threads increases. Usually, there should be linear correlation between the FPGA performance and the number of threads. But as Fig. 6 shows, the performance of the 16 threads design behaves abnormally because of decrease in the frequency from 250MHz to 230MHz. Although major FPGA on-chip resources are far from completely consumed (2364480 FF, 1182240 LUT, 6840 DSP48E, 4320 BRAM_18K total available), the rapidly growing delay and shortage of routing resources prevent the number of thread from continuing to increase. Currently, the number of threads is set 8 in the FPGA implementation.

We compare the FPGA/CPU heterogeneous implementation with the CPU implementation (Table III). In test case 1, the speedup ratio of performance is 6.09x and the speedup ratio of energy efficiency is 42.63x. In test case 2, the speedup ratio of performance is 4.92x and the speedup ratio of energy efficiency is 37.58x.

The FRW algorithm in this paper is the standard version of WOS. Implementations in [8] and [9] are optimized version of WOC, which is very sophisticated. As illustrated in Table IV, our FPGA/CPU heterogeneous implementation still has

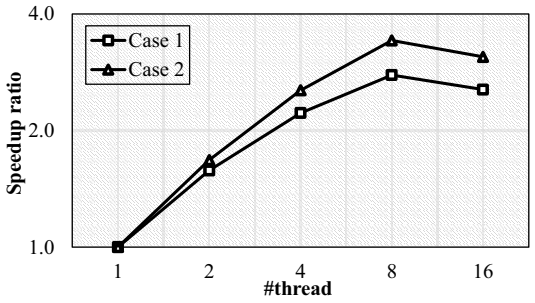

Fig. 6. Relationship between FPGA performance and $\#thread$ (log-log scale)

TABLE III
COMPARISON OF FPGA/CPU HETEROGENEOUS IMPLEMENTATION AND CPU IMPLEMENTATION

Case	*Platform*	*#thread*	*Performance (paths/s)*	*Power (W)*	*Energy efficiency (paths/s/W)*	*SU[1]*	*SU[2]*
1	CPU	4	3842.31	84	45.74	-	-
1	FPGA+CPU	8	23400.11	12	1950.01	6.09	42.63
2	CPU	4	4055.64	84	48.28	-	-
2	FPGA+CPU	8	19957.99	11	1814.36	4.92	37.58

[1] Speedup of performance.
[2] Speedup of energy efficiency.

TABLE IV
FPGA/CPU HETEROGENEOUS IMPLEMENTATION STILL HAS AN ADVANTAGE ON ENERGY EFFICIENCY OVER THE CPU IMPLEMENTATION OF THE OPTIMIZED WOC

Platform	*Algorithm*	*Performance (paths/s)*	*Power (W)*	*Energy efficiency (paths/s/W)*
Intel Core i5-4570	standard	3842.31	84	45.74
Intel Xeon E5620 [8]	optimized	29931.97	80	374.15
NVIDIA GTX 580 [9]	optimized	1691666.67	244	6933.06
NVIDIA GTX 580 [9]	standard	662921.35	244	2716.89
FPGA+CPU[1]	standard	23400.11	12	1950.01

[1] Xilinx VU9P + Intel Xeon E5-2686.

5.2x energy efficiency over the implementation of the optimized WOC on an 8-core CPU. Considering our framework is scalable, there is great potential of improvement in its performance, only with multiple FPGAs, or just an FPGA that has more routing resources.

V. CONCLUSIONS

In this paper, we present a scalable FPGA/CPU heterogeneous framework of the FRW algorithm with WOS. The challenges of developing FRW on an FPGA are limited on-chip resource and non-intuitive kernel design. In the proposed framework, a CPU takes charge of partitioning input circuits into segments of appropriate scale, and the random walking is processed by an FPGA with bitstream built once for all. FPGA kernel optimization techniques like pipes, loop unrolling, loop pipelining and array partition are also applied to extract maximum parallelism of the FPGA. The design has been implemented on AWS EC2 F1 (Xilinx VU9P FPGA board) with Xilinx SDAccel toolchain. The implementation shows 5x performance and 40x energy efficiency to the CPU solution, and 5x energy efficiency over the state-of-the-art WOC implementation on an 8-core CPU. Thanks to the scalability of our framework, further improvement can be easily achieved with multiple FPGAs, or just an FPGA that has more routing resources.

ACKNOWLEDGMENT

This work is supported partly by the National Major Science and Technology Special Project of China (2017ZX01028101-003); partly by National Natural Science Foundation of China (NSFC) research projects 61674042, 61574046, 61774045, 61574044 and 61628402; and partly by National Science Foundation (NSF) under CCF-1533656.

REFERENCES

[1] K. Nabors and J. White, "Fastcap: A multipole accelerated 3-d capacitance extraction program," *IEEE Trans. on CAD*, vol. 10, no. 11, pp. 1447–1459, 1991.
[2] W. Shi, J. Liu, N. Kakani, and T. Yu, "A fast hierarchical algorithm for three-dimensional capacitance extraction," *IEEE Trans. on CAD*, vol. 21, no. 3, pp. 330–336, 2002.
[3] W. Chai, D. Jiao, and C.-K. Koh, "A direct integral-equation solver of linear complexity for large-scale 3d capacitance and impedance extraction," in *DAC*. IEEE, 2009, pp. 752–757.
[4] N. Van der Meijs and A. J. van Genderen, "An efficient finite element method for submicron ic capacitance extraction," in *DAC*. ACM, 1989, pp. 678–681.
[5] G. Chen, H. Zhu, T. Cui, Z. Chen, X. Zeng, and W. Cai, "Parafemcap: a parallel adaptive finite-element method for 3-d vlsi interconnect capacitance extraction," *IEEE Trans. on MTT*, vol. 60, no. 2, pp. 218–231, 2012.
[6] Y. Le Coz and R. Iverson, "A stochastic algorithm for high speed capacitance extraction in integrated circuits," *Solid-State Electronics*, vol. 35, no. 7, pp. 1005–1012, 1992.
[7] N. Sawhney, S. Batterywala, N. Shenoy, and R. Rudell, "Parallelizing a statistical capacitance extractor," *VDAT*, pp. 253–267, 2004.
[8] W. Yu, H. Zhuang, C. Zhang, G. Hu, and Z. Liu, "Rwcap: A floating random walk solver for 3-d capacitance extraction of very-large-scale integration interconnects," *IEEE Trans. on CAD*, vol. 32, no. 3, pp. 353–366, 2013.
[9] K. Zhai, W. Yu, and H. Zhuang, "Gpu-friendly floating random walk algorithm for capacitance extraction of vlsi interconnects," in *DATE*. EDA Consortium, 2013, pp. 1661–1666.
[10] N. D. Arora, S. Worley, and D. R. Ganpule, "Fieldrc, a gpu accelerated interconnect rc parasitic extractor for full-chip designs," in *EDSSC*. IEEE, 2015, pp. 459–462.
[11] L. Wirbel, "Xilinx sdaccel whitepaper," 2014.
[12] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks," in *ICCAD*. ACM, 2016, p. 12.
[13] E. Del Sozzo, L. Di Tucci, and M. D. Santambrogio, "A highly scalable and efficient parallel design of n-body simulation on fpga," in *IPDPSW*. IEEE, 2017, pp. 241–246.
[14] V. Dang and K. Skadron, "Acceleration of frequent itemset mining on fpga using sdaccel and vivado hls," in *ASAP*. IEEE, 2017, pp. 195–200.
[15] T. Kenter, J. Förstner, and C. Plessl, "Flexible fpga design for fdtd using opencl," in *FPL*. IEEE, 2017, pp. 1–7.